

IF2211-03 STRATEGI ALGORITMA

Pengaplikasian Algoritma BFS dan DFS dalam Menyelesaikan Persoalan Maze Treasure Hunt

LAPORAN TUGAS BESAR 2



Oleh :

Kelompok 1 - besokKelar

Henry Anand Septian Radityo - 13521004

Bernardus Willson - 13521021

Kenny Benaya Nathan - 13521023

**Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2022**

DAFTAR ISI

PENDAHULUAN	4
BAB II	6
LANDASAN TEORI	6
2.1. Traversal Graf	6
2.2. Breadth-First Search (BFS)	7
2.3. Depth-First Search (DFS)	7
2.4. C# Application Desktop Development	8
BAB III	10
APLIKASI STRATEGI BFS DAN DFS	10
3.1. Langkah Pemecahan Masalah	10
3.2. Algoritma Breadth First Search	10
3.3. Algoritma Depth First Search	11
3.4. Algoritma Traveling Salesman Problem	11
3.4.1. Algoritma TSP dengan BFS	12
3.4.2. Algoritma TSP dengan DFS	12
3.5. Ilustrasi	13
3.5.1. Algoritma BFS	13
3.5.2. Algoritma DFS	15
BAB IV	17
ANALISIS PEMECAHAN MASALAH	17
4.1. Implementasi Program	17
4.2.1. Pseudocode Simpul	17
4.2.2. Pseudocode Breadth First Search	19
4.2.3. PseudoCode DepthFirst Search	24
4.2. Struktur Data	28
4.2.1. Kelas Simpul	28
4.2.2. Kelas BFS	29
4.2.3. Kelas DFS	31
4.3. Cara Penggunaan Program	33
4.6. Hasil Pengujian	35
4.6.1. Pencarian rute BFS Test Case 1	35
4.6.2. Pencarian rute DFS Test Case 1	36
4.6.3. Pencarian rute BFS Test Case 2	36
4.6.4. Pencarian rute DFS Test Case 2	37
4.6.5. Pencarian rute BFS/DFS Test Case 3	37
4.6.6. Pencarian rute BFS Test Case 4	38
4.6.7. Pencarian rute DFS Test Case 4	38

4.6.8. Pencarian rute BFS Test Case 5	39
4.6.9. Pencarian rute DFS Test Case 5	39
4.6.10. Pencarian rute BFS TSP Test Case 1 (BONUS)	40
4.6.11. Pencarian rute DFS TSP Test Case 1 (BONUS)	40
4.7. Analisis	41
BAB V	42
KESIMPULAN DAN SARAN	42
5.1. Kesimpulan	42
5.2. Saran	42
5.3. Refleksi	42
5.4. Tanggapan	43
REFERENSI	44
TAUTAN	45

BAB I

PENDAHULUAN

Tuan Krabs menemukan sebuah labirin distorsi terletak tepat di bawah Krusty Krab bernama El Doremi yang Ia yakini mempunyai sejumlah harta karun di dalamnya dan tentu saja Ia ingin mengambil harta karunnya. Dikarenakan labirinnya dapat mengalami distorsi, Tuan Krabs harus terus mengukur ukuran dari labirin tersebut. Oleh karena itu, Tuan Krabs banyak menghabiskan tenaga untuk melakukan hal tersebut sehingga Ia perlu memikirkan bagaimana caranya agar Ia dapat menelusuri labirin ini lalu memperoleh seluruh harta karun dengan mudah.

Setelah berpikir cukup lama, Tuan Krabs tiba-tiba mengingat bahwa ketika Ia berada pada kelas Strategi Algoritma-nya dulu, Ia ingat bahwa Ia dulu mempelajari algoritma BFS dan DFS sehingga Tuan Krabs menjadi yakin bahwa persoalan ini dapat diselesaikan menggunakan kedua algoritma tersebut. Akan tetapi, dikarenakan sudah lama tidak menyentuh algoritma, Tuan Krabs telah lupa bagaimana cara untuk menyelesaikan persoalan ini dan Tuan Krabs pun kebingungan. Tidak butuh waktu lama, Ia terpikirkan sebuah solusi yang brilian. Solusi tersebut adalah meminta mahasiswa yang saat ini sedang berada pada kelas Strategi Algoritma untuk menyelesaikan permasalahan ini.

Dalam tugas besar ini, Anda akan diminta untuk membangun sebuah aplikasi dengan GUI sederhana yang dapat mengimplementasikan BFS dan DFS untuk mendapatkan rute memperoleh seluruh treasure atau harta karun yang ada. Program dapat menerima dan membaca input sebuah file txt yang berisi maze yang akan ditemukan solusi rute mendapatkan treasure-nya. Untuk mempermudah, batasan dari input maze cukup berbentuk segi-empat dengan spesifikasi simbol sebagai berikut :

- K : Krusty Krab (Titik awal)
- T : Treasure
- R : Grid yang mungkin diakses / sebuah lintasan
- X : Grid halangan yang tidak dapat diakses

Dengan memanfaatkan algoritma Breadth First Search (BFS) dan Depth First Search (DFS), anda dapat menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute

solusi adalah rute yang memperoleh seluruh treasure pada maze. Perhatikan bahwa rute yang diperoleh dengan algoritma BFS dan DFS dapat berbeda, dan banyak langkah yang dibutuhkan pun menjadi berbeda. Prioritas arah simpul yang dibangkitkan dibebaskan asalkan ditulis di laporan ataupun readme, semisal LRUD (left right up down). Tidak ada pergerakan secara diagonal. Anda juga diminta untuk memvisualisasikan input txt tersebut menjadi suatu grid maze serta hasil pencarian rute solusinya. Cara visualisasi grid dibebaskan, sebagai contoh dalam bentuk matriks yang ditampilkan dalam GUI dengan keterangan berupa teks atau warna. Pemilihan warna dan maknanya dibebaskan ke masing-masing kelompok, asalkan dijelaskan di readme / laporan.

Daftar input maze akan dikemas dalam sebuah folder yang dinamakan test dan terkandung dalam repository program. Folder tersebut akan setara kedudukannya dengan folder src dan doc (struktur folder repository akan dijelaskan lebih lanjut di bagian bawah spesifikasi tubes). Cara input maze boleh langsung input file atau dengan textfield sehingga pengguna dapat mengetik nama maze yang diinginkan. Apabila dengan textfield, harus menghandle kasus apabila tidak ditemukan dengan nama file tersebut.

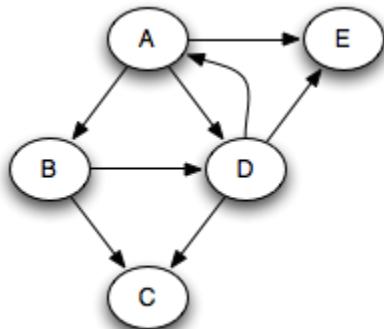
Setelah program melakukan pembacaan input, program akan memvisualisasikan gridnya terlebih dahulu tanpa pemberian rute solusi. Hal tersebut dilakukan agar pengguna dapat mengerjakan terlebih dahulu treasure hunt secara manual jika diinginkan. Kemudian, program menyediakan tombol solve untuk mengeksekusi algoritma DFS dan BFS. Setelah tombol diklik, program akan melakukan pemberian warna pada rute solusi.

BAB II

LANDASAN TEORI

2.1. Traversal Graf

Graf adalah sebuah representasi dari berbagai objek yang terpisah bersama dengan hubungan yang ada antar objek. Objek tersebut dapat disebut sebagai simpul (*node*), sementara hubungan antar objek yang biasa digambarkan sebagai sebuah garis antara simpul dapat disebut sebagai sisi (*edge*).



Gambar 1 Graf

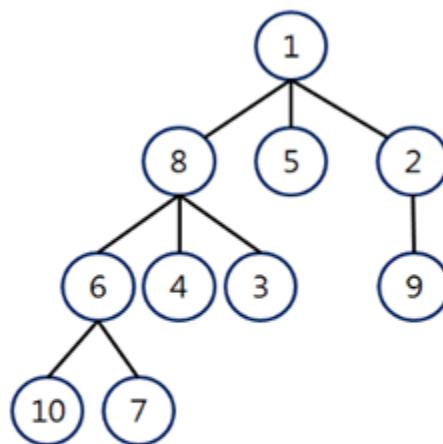
(sumber: www.cs.cornell.edu/courses/cs2112/2012sp/lectures/lec24/lec24-12sp.html)

Algoritma traversal graf adalah proses mengunjungi simpul-simpul yang terdapat dalam graf secara sistematik. Di sini, graf dapat digambarkan menjadi sebuah persoalan yang ingin dicari solusinya dengan melakukan traversal graf. Untuk mengunjungi simpul-simpul ini, kita tidak dapat melakukannya dengan asal mengunjungi simpul yang terhubung (tetangga) dengan suatu simpul karena banyak graf memiliki *loop* (simpul yang terhubung dengan simpul itu sendiri). Oleh karena itu, kita harus memiliki catatan simpul; bisa dalam bentuk boolean; yang sudah dikunjungi dengan perencanaan yang baik untuk rute sisi yang akan dikunjungi selanjutnya agar tidak bergerak berulang pada simpul yang sama.

Ada dua pendekatan umum yang termasuk dalam algoritma traversal graf, yaitu *Breadth-First Search* (BFS) dan *Depth-First Search* (DFS). Kedua algoritma ini termasuk dalam *uninformed/blind search*. Hal ini menunjukkan bahwa graf yang menjadi persoalan tidak memberikan informasi lain selain hubungan/*edge* tersebut. Oleh karena itulah kedua algoritma bergerak secara *brute-force* sehingga melakukan *blind searching*.

2.2. Breadth-First Search (BFS)

Pencarian melebar atau BFS merupakan sebuah algoritma di mana pencarian dilakukan dengan mengunjungi seluruh tetangga yang terhubung langsung dengan simpul yang sedang diacu (*search key*). Seluruh tetangga yang sedang dikunjungi akan dimasukkan ke sebuah *queue* yang memanfaatkan prinsip *First-In First-Out* (FIFO) sebagai urutan untuk *search key* selanjutnya. Apabila sebuah simpul akan dijadikan *search key*, simpul tersebut akan di-*dequeue* dari *queue* yang ada. Ketika seluruh tetangganya sudah dikunjungi, *search key* akan dipindah ke level selanjutnya atau kepada simpul yang satu level dengan *search key* sebelumnya, tergantung dengan urutan *queue*.



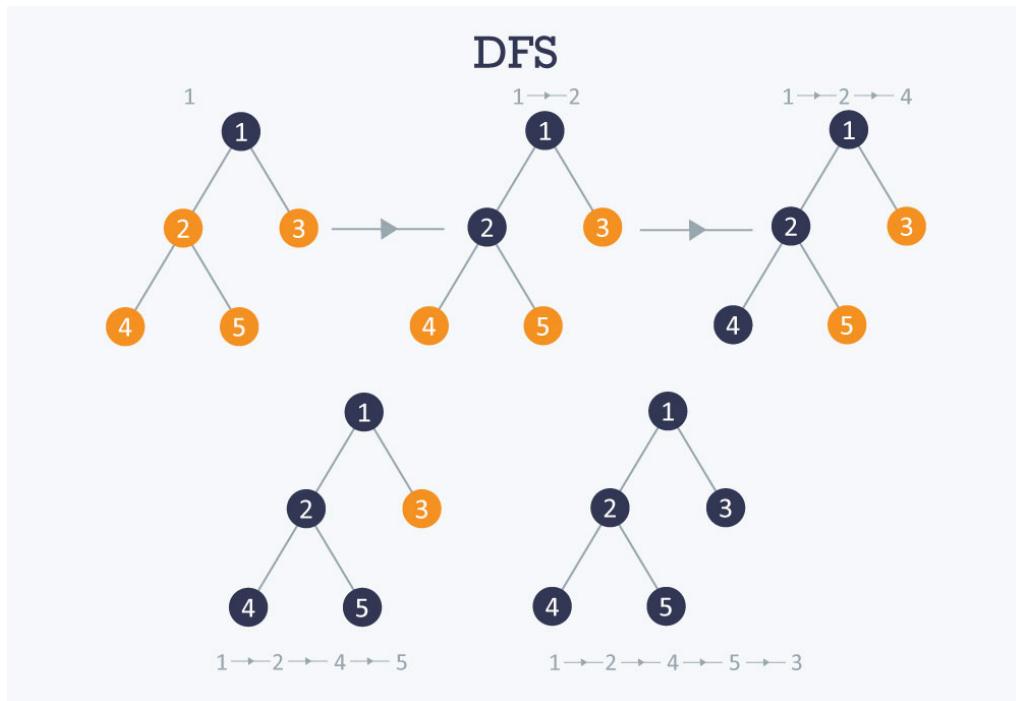
BFS: 1 8 5 2 6 4 3 9 10 7

Gambar 2 Contoh penerapan BFS pada sebuah graf
(sumber: www.freecodecamp.org/news/breadth-first-search-a-bfs-graph-traversal-guide-with-3-leetcodeexamples/)

2.3. Depth-First Search (DFS)

Pencarian mendalam atau DFS merupakan sebuah algoritma di mana pencarian dilakukan dengan mengunjungi salah satu tetangga dari *search key*. Cara ini dilakukan berulang kali hingga menyentuh simpul terdalam atau simpul yang sudah tidak memiliki tetangga lagi (atau seluruh tetangganya sudah dikunjungi). Apabila sebuah simpul akan dijadikan *search key*, maka simpul tersebut akan di-*pop* dari *stack* yang ada. Seluruh tetangga yang belum dikunjungi dari *search key* akan dimasukkan ke dalam sebuah *stack* yang memanfaatkan prinsip *Last-In First-Out*

(LIFO). Ketika sudah mencapai simpul terdalam, algoritma ini akan menerapkan algoritma *backtracking* di mana *search key* akan menelusuri kembali simpul sebelumnya yang sudah dikunjungi dan masih memiliki tetangga yang belum dikunjungi. Dalam hal ini, simpul yang ditelusuri sudah terurut dari *stack* yang ada.



Gambar 3 Contoh penerapan DFS pada sebuah graf
(sumber: www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial)

2.4. C# Application Desktop Development

C# (C-sharp) adalah bahasa pemrograman yang dikembangkan oleh Microsoft dan digunakan untuk membangun aplikasi desktop, web, dan mobile. C# Application Desktop Development adalah pengembangan aplikasi desktop menggunakan bahasa pemrograman C# dan platform .NET Framework.

Dalam pengembangan aplikasi desktop dengan C#, developer biasanya menggunakan Integrated Development Environment (IDE) seperti Microsoft Visual Studio untuk memudahkan proses pembuatan aplikasi. Dalam IDE ini, developer dapat membuat antarmuka pengguna (user interface) aplikasi dengan drag-and-drop, menambahkan kode program untuk menghubungkan antarmuka dengan data dan logika aplikasi, serta melakukan debugging untuk menemukan dan memperbaiki kesalahan dalam kode program.

Untuk membuat aplikasi desktop yang berfungsi dengan baik, developer C# harus memahami konsep-konsep dasar seperti tipe data, variabel, operator, aliran kontrol, dan fungsi. Selain itu, mereka juga perlu mempelajari bagaimana menggunakan API .NET Framework untuk mengakses dan memanipulasi data dan sumber daya lainnya yang terkait dengan aplikasi desktop.

Pengembangan aplikasi desktop dengan C# dan .NET Framework memiliki keuntungan dalam hal keamanan dan stabilitas aplikasi, serta ketersediaan library dan tools yang mendukung pengembangan aplikasi dengan cepat dan efisien. Aplikasi desktop yang dikembangkan dengan C# juga dapat berjalan pada berbagai sistem operasi, termasuk Windows, macOS, dan Linux.

BAB III

APLIKASI STRATEGI BFS DAN DFS

3.1. Langkah Pemecahan Masalah

Langkah awal yang digunakan penulis untuk mendapatkan semua treasure yang tersedia adalah dengan merekonstruksi ulang maze yang ada menjadi matriks yang berisikan elemen-elemen integer. Hal ini bertujuan agar pengecekan tiap tile pada maze dapat lebih mudah. Adapun perubahan simbol menjadi integer sebagai berikut.

$$\begin{aligned} K &\rightarrow 1 \\ R &\rightarrow 2 \\ T &\rightarrow 3 \\ X &\rightarrow 0 \end{aligned}$$

Setelah maze yang dimasukkan diubah menjadi sebuah matriks, maka akan dimulai pencarian untuk titik start dari maze. Untuk itu dilakukan proses searching secara traversal pada matriks dengan tujuan menemukan elemen dengan angka satu. Setelah ditemukan titik awal dari maze dan juga maze yang sudah berubah dalam bentuk matriks, selanjutnya akan dilakukan proses pencarian treasure dengan menggunakan algoritma Breadth First Search dan Depth First Search.

3.2. Algoritma Breadth First Search

Algoritma BFS (*Breadth First Search*) merupakan algoritma yang digunakan untuk menemukan jalur terpendek dalam sebuah graf atau *maze*. Dalam algoritma BFS setiap kotak dalam *maze* akan diibaratkan seperti sebuah simpul dan simpul yang bersebelahan secara vertikal dan horizontal akan dapat diakses oleh simpul tersebut. Dalam kasus ini, simpul yang memiliki tanda X tidak akan dapat diakses karena menunjukkan tembok. Secara umum algoritma BFS untuk mendapatkan jalur dalam mengambil semua treasure akan melalui beberapa tahapan.

1. Dibuat *queue* kosong untuk menampung kumpulan rute yang mungkin dalam tiap *maze*
2. Koordinat dari titik Start akan di-*enqueue* ke dalam *queue*
3. *Queue* akan di-*dequeue* lalu dicari kemungkinan *node* yang mungkin untuk dikunjungi (kanan-bawah-kiri-atas), apabila terdapat *node* yang mungkin untuk dikunjungi maka akan di-*enqueue* kembali ke dalam *queue*
4. Proses nomor 3 akan diulang hingga menemukan sebuah treasure.

5. Apabila treasure sudah ditemukan, maka queue akan di clear terlebih dahulu lalu menjadikan titik treasure yang ditemukan sebagai titik start.
6. Langkah diatas akan terus diulang hingga mendapatkan semua treasure.
7. Kumpulan path yang menuju treasure akan digabungkan sehingga membentuk suatu jalur yang mengunjungi semua treasure.

3.3. Algoritma Depth First Search

Algoritma Depth-First Search (DFS) adalah salah satu algoritma yang dapat digunakan untuk menyelesaikan masalah maze atau labirin. DFS bekerja dengan mengunjungi simpul secara berurutan hingga mencapai solusi atau tidak ada simpul yang dapat dikunjungi lagi. Sama seperti BFS, algoritma ini akan dilakukan untuk matriks maze dan titik koordinat awal sebagai titik start. Dalam pengaplikasiannya, terdapat beberapa tahap untuk mencari rute dalam mengambil semua treasure yang ada pada maze.

1. Dibuat dua stack A dan B, keduanya akan diisi dengan koordinat awal dari maze. Stack A akan digunakan untuk mencatat rute secara keseluruhan sedangkan stack B akan digunakan untuk melakukan pencarian atau searching.
2. Dibuat sebuah map yang digunakan untuk menandai rute yang telah dilewati. Titik yang telah dilewati akan ditandai dengan integer -1.
3. Diambil top dari stack B dan dicari kemungkinan untuk bergerak mulai dari prioritas kanan-bawah-kiri-atas. Rute yang ingin dilewati juga harus dicek apakah sudah pernah dilewati atau belum, karena algoritma hanya akan melakukan pengecekan pada rute yang belum dilewati.
4. Apabila terdapat koordinat yang dapat dilewati, maka koordinat tersebut akan dipush ke kedua stack A dan B.
5. Apabila terdapat jalan buntu, maka akan dilakukan pop pada stack B hingga menemukan suatu koordinat yang memiliki kemungkinan untuk bergerak.
6. Algoritma akan terus dilakukan hingga mendapatkan semua treasure yang ada di dalam maze.

3.4. Algoritma Traveling Salesman Problem

Pada program ini, pengembang juga menyisipkan semacam algoritma seperti *Travelling Salesman Problem* (TSP) yang bertujuan untuk mencari jalan untuk kembali ke tempat awal.

Algoritma yang digunakan memanfaatkan masing-masing algoritma BFS dan DFS untuk penyelesaiannya. Tidak seperti algoritma TSP pada umumnya, tujuan dari penggunaan algoritma ini adalah hanya untuk kembali ke tempat awal tanpa memperhatikan sirkuit hamilton terpendek. Jadi, penyelesaian masalah disini tidak dilakukan dengan melakukan optimasi, namun dilakukan secara traversal untuk mencari jalur untuk kembali ke tempat awal seefektif mungkin dan tidak harus melewati seluruh *node* (jalur) yang tersedia.

3.4.1. Algoritma TSP dengan BFS

Ketika sedang menyelesaikan *maze* dengan algoritma BFS, maka program juga akan menjalankan program BFS untuk kembali ke tempat semula. Secara umum, algoritma BFS yang dijalankan untuk kembali ke titik awal akan melalui beberapa tahapan lagi dan tahapan ini sangat mirip, bahkan hampir sama dengan tahapan ketika mencari seluruh *treasure* dalam *maze*. Tahapan tersebut mencakup:

1. Setelah mendapatkan *treasure* terakhir, program akan membuat *queue* baru lagi yang kosong untuk menampung kumpulan rute
2. Koordinat dari titik *treasure* terakhir yang ditemukan akan di-*enqueue* ke dalam *queue*
3. Queue akan di-*dequeue* lalu dicari kemungkinan *node* yang mungkin untuk dikunjungi (kanan-bawah-kiri-atas), apabila terdapat *node* yang mungkin untuk dikunjungi maka akan di-*enqueue* kembali ke dalam *queue*
4. Proses nomor 3 akan diulang hingga akhirnya kembali lagi ke titik awal
5. Ketika sudah mencapai titik awal, maka *path* menuju titik awal ini akan digabungkan dengan *path* dari titik awal menuju ke seluruh *treasure*, sehingga akan membentuk jalur dari titik awal ke seluruh *treasure* sampai kembali lagi ke titik awal

3.4.2. Algoritma TSP dengan DFS

Sama seperti algoritma TSP dengan BFS, ketika sedang menyelesaikan *maze* dengan algoritma DFS, maka program juga akan menjalankan program DFS untuk kembali ke tempat semula dan tahapannya juga hampir sama dengan tahapan algoritma ketika mencari seluruh *treasure* dalam *maze*. Secara umum, algoritma DFS yang diaplikasikan untuk kembali ke titik awal akan melalui beberapa tahapan lagi yang terdiri dari:

- Setelah mendapatkan *treasure* terakhir, program akan membuat sebuah stack baru (sebut saja stack C) yang akan melakukan pencarian khusus kembali ke titik awal. Sementara itu, stack A akan tetap dipakai melanjutkan untuk tahapan ini
- Dibuat kembali suatu map yang digunakan untuk menandai rute yang telah dilewati, namun kali ini titik yang sudah dilewati akan ditandai dengan integer -2.
- Koordinat dari *treasure* akhir akan dimasukkan ke dalam Stack A (yang dipakai sebelumnya ketika mencari seluruh *treasure*) dan stack C.
- Top dari stack B akan diambil untuk mencari kemungkinan untuk bergerak dengan prioritas yang sama dengan tahap sebelumnya, yaitu kanan-bawah-kiri-atas. Rute yang ingin dilewati juga harus dicek kembali apakah sudah dilewati atau belum, karena algoritma juga mengecek jalur yang belum dilewati
- Apabila ada jalur yang bisa dilewati, koordinat tersebut akan di-push ke dalam kedua stack A dan C
- Apabila terdapat jalan buntu, maka akan dilakukan *pop* dari stack C sampai menemukan sebuah koordinat yang memiliki cabang yang bisa dilewati
- Algoritma ini akan terus diulang hingga kembali ke titik awal

3.5. Ilustrasi

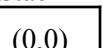
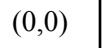
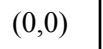
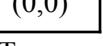
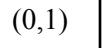
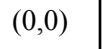
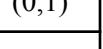
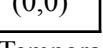
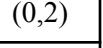
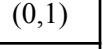
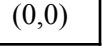
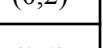
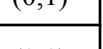
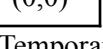
Coordinate	0	1	2
0	Start		Treasure
1		Treasure	

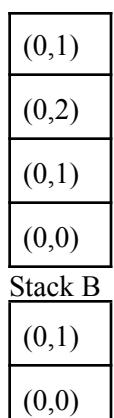
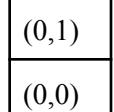
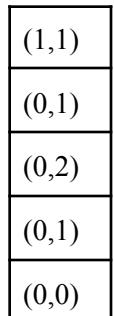
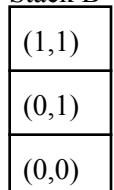
3.5.1. Algoritma BFS

Queue	Keterangan
{[(0,0)]} Temporary Path : - Result : {}	Titik awal dari start di enqueue
{} Temporary Path : [(0,0)] Result : {}	Dequeue Dilakukan pengecekan untuk node yang dapat dijangkau oleh temporary path, dalam kasus ini yang mungkin adalah menuju koordinat (0,1)

<p>$\{(0,0)(0,1)\}$ Temporary Path : - Result : {}</p>	Temporary akan diupdate sesuai dengan simpul yang mungkin dijangkau lalu dienqueue kembali
<p>{} Temporary Path : $[(0,0)(0,1)]$ Result : {}</p>	<p>Dequeue Karena pada titik 0,1 kemungkinan jalur dapat menuju ke kiri kanan dan atas maka akan dilakukan update pada temporary untuk koordinat tersebut.</p>
<p>$\{(0,0)(0,1)(0,2)\},$ $[(0,0)(0,1)(1,1)],$ $[(0,0)(0,1)(0,0)]\}$ Temporary Path : - Result : {}</p>	<p>Dilakukan update pada temporary lalu di enqueue ke dalam queue. Karena titik (0,2) telah ditemukan treasure maka jalur tersebut disimpan dan kemudian queue akan dibersihkan. Start akan dimulai dari titik (0,2). Treasure pada titik (0,2) akan dihilangkan karena sudah terambil.</p>
<p>$\{(0,2)\}$ Temporary Path : - Result : $\{[(0,0)(0,1)(0,2)]\}$</p>	Lakukan enqueue pada titik start
<p>{} Temporary Path : $[(0,2)]$ Result : $\{[(0,0)(0,1)(0,2)]\}$</p>	<p>Dequeue Karena kemungkinan hanya ke kiri maka lakukan update untuk koordinat tersebut</p>
<p>$\{(0,2)(0,1)\}$ Temporary Path : - Result : $\{[(0,0)(0,1)(0,2)]\}$</p>	Lakukan enqueue untuk temporary path yang sudah terupdate
<p>{} Temporary Path : $[(1,1)(0,1)]$ Result : $\{[(0,0)(0,1)(0,2)]\}$</p>	<p>Lakukan dequeue, karena pada titik 0,1 terdapat kemungkinan untuk ke kanan, bawah dan kiri, maka akan dilakukan update terhadap temporary path.</p>
<p>$\{(0,2)(0,1)(0,2)\},$ $[(0,2)(0,1)(1,1)],$ $[(0,2)(0,1)(0,0)]\}$ Temporary Path : - Result : $\{[(0,0)(0,1)(0,2)]\}$</p>	<p>Dilakukan update pada temporary lalu di enqueue ke dalam queue. Karena titik (1,1) telah ditemukan treasure maka jalur tersebut disimpan dan kemudian queue akan dibersihkan. Kemudian jalur akan disimpan.</p>
<p>Result : $\{[(0,0)(0,1)(0,2)], [(0,2)(0,1)(1,1)]\}$</p>	<p>Dengan demikian jalur untuk mengambil treasure sudah didapatkan. Kemudian akan dilakukan merge agar titik (0,2) tidak terulang.</p>

3.5.2. Algoritma DFS

Stack	Keterangan
Stack A  Stack B  Temporary Coordinate : -	Dilakukan push untuk koordinat start pada stack A dan Stack B. Titik (0,0) akan ditandai sebagai titik yang pernah dilalui.
Stack A  Stack B  Temporary Coordinate : (0,0)	Top dari stack B akan dicek untuk mencari kemungkinan arah gerak Prioritas : kanan-bawah-kiri-atas Karena kanan dapat dilakukan maka akan dilakukan push untuk koordinat (0,1), dan titik tersebut akan ditandai sebagai titik yang pernah dilalui
Stack A   Stack B   Temporary Coordinate : (0,1)	Dilakukan push untuk koordinat yang mungkin lalu temp akan berubah menjadi top dari stack B yang baru. Kemudian akan dicari kembali kemungkinan gerak sesuai dengan urutan prioritas, sesuai dengan prioritas, titik yang dapat dikunjungi adalah titik (0,2). Maka akan dilakukan push dan (0,2) akan ditandai sebagai titik yang pernah dikunjungi. Lalu karena (0,2) memiliki treasure, count untuk treasure akan bertambah.
Stack A    Stack B    Temporary Coordinate : (0,2)	Di titik (0,2) tidak ada lagi kemungkinan untuk bergerak karena pada titik (0,1) sudah pernah dikunjungi sedangkan arah lainnya buntu. Oleh karena itu akan dilakukan pop untuk Stack B sampai ditemukan koordinat yang masih mungkin untuk bergerak dan push pada stack A. Dalam hal ini akan dilakukan pop hingga koordinat (0,1).
Stack A	Di titik (0,1) jalur yang mungkin untuk dilewati hanya ke bawah. Oleh karena itu akan dilakukan

 <p>Stack B</p>  <p>Temporary Coordinate : (0,1)</p>	<p>push (1,1). Titik (1,1) akan ditandai karena sudah dilewati dan karena titik tersebut memiliki treasure, maka cout of treasure bertambah.</p>
<p>Stack A</p>  <p>Stack B</p>  <p>Temporary Coordinate : (1,1)</p>	<p>Karena count of treasure sudah sama dengan total treasure maka program akan berhenti dan jalur akan ada pada stack A.</p>

BAB IV

ANALISIS PEMECAHAN MASALAH

4.1. Implementasi Program

4.2.1. Pseudocode Simpul

```
Simpul:  
    x: int  
    y: int  
    canGoLeft: bool  
    canGoRight: bool  
    canGoDown: bool  
    canGoUp: bool  
    maze: int[,]  
    arr: List of (int, int)  
    idx: int  
  
Simpul(x: int, y: int, maze: int[,]):  
    x ← x  
    y ← y  
    maze ← maze  
    create a new List of Tuples and set it to arr  
    set idx to -1  
  
    if (y+1 < maze.GetLength(1)) then  
        if (maze[x, y+1] != 0) then  
            canGoRight ← true  
        else  
            canGoRight ← false  
    else  
        canGoRight ← false  
  
    if (y - 1 >= 0) then  
        if (maze[x, y-1] != 0) then  
            canGoLeft ← true  
        else  
            canGoLeft ← false  
    else  
        canGoLeft ← false  
  
    if (x - 1 >= 0) then  
        if (maze[x-1, y] != 0) then  
            canGoUp ← true  
        else  
            canGoUp ← false  
    else  
        canGoUp ← false  
  
    if (x+1 < maze.GetLength(0)) then  
        if (maze[x+1, y] != 0) then  
            canGoDown ← true  
        else  
            canGoDown ← false
```

```

else
    canGoDown ← false

Simpul(other: Simpul):
    x ← other.x
    y ← other.y
    maze ← other.maze
    canGoDown ← other.canGoDown
    canGoLeft ← other.canGoLeft
    canGoRight ← other.canGoRight
    canGoUp ← other.canGoUp
    create a new List of Tuples, copy other.arr to arr
    idx ← other.idx

displayMaze():
    for i traversal [0 .. maze.GetLength(0) - 1]
        for j traversal [0 .. maze.GetLength(1) - 1]:
            Output (maze[i,j], " ")
            output (\n)

isVisited(x: int, y: int) -> bool:
    if (maze[x,y] == -1) then
        → true
    else
        → false

isVisitedHome(x: int, y: int) -> bool:
    if (maze[x,y] == -2) then
        → true
    else
        → false

visit(x: int, y: int):
    maze[x,y] ← -1

visitHome(x: int, y: int):
    maze[x,y] ← -2

getArr() -> List of (int, int):
    → arr

Append(x: int, y: int):
    add a new Tuple of (x,y) to arr
    Idx ← idx + 1

function unite(Queue<Simpul> p):
    bool first ← true
    for each Simpul value in p:
        if (first) then
            first ← false
            for each (int x, int y) in value.getArr():
                arr.add((x,y))
        else:
            for i from 1 to value.getArr().Count():
                (int x, int y) = value.getArr()[i]

```

```

        arr.add( (x,y) )

function displayCoord():
    for each (int x, int y) in arr:
        output("(" + x + " " + y + ")")

function isTreasure(int x, int y):
    if maze[x,y] == 3 then
        → true
    else:
        → false

function isHome(int x, int y):
    if maze[x,y] == 1 then
        → true
    else:
        → false

function getIdx():
    → idx

function getX():
    → x

function getY():
    → y

function getEl(int id):
    int[] ret ← new int[2]
    (int x, int y) ← arr[id]
    ret[0] ← x
    ret[1] ← y
    → ret

function displaySimpul():
    output(x + " " + y)

function getMaze():
    → maze

function pickTreasure(int x, int y):
    maze[x,y] ← 2

```

4.2.2. Pseudocode Breadth First Search

```

BFS:
path: queue
flag: bool
coorMap: queue
result: queue
result: progress
progress: int
tempProgress: Simpul
x: int

```

```

y: int
maze: arr of (int,int)
visitedMaze: arr of (int,int)
count : int
countSteps : int
countVisited : int
goHome : bool
BFS(int x, int y, int[,] maze, int total):
    initialize coorMap as an empty queue
    initialize result as an empty queue
    initialize progress as an empty queue
    initialize tempProgress as a Simpul
    x ← x
    Y ← Y
    maze ← copy of maze
    visitedMaze ← copy of maze
    count ← total
    countSteps ← 0
    countVisited ← 0
    goHome ← false

canTravel(int x, int y):
    if (x≥0 && x<maze.GetLength(0) && y≥0 && y<maze.GetLength(1)) then
        if (maze[x,y] != 0) then
            → true
        else
            → false
    else
        → false

initHome():
    initialize coorMap as an empty queue
    found ← false

    if (!tempProgress.isVisitedHome(x,y)) then
        tempProgress.Append(x,y)
        tempProgress.visitHome(x,y)

    if (this.y+1 < this.maze.GetLength(1) && this.maze[x,y+1] != 0 && ! found)
then
        temp ← new Simpul(x,y,this.maze)
        temp.Append(x,y)
        enqueueResultInitHome(x, y + 1, ref temp, ref found)

    if (this.x+1 < this.maze.GetLength(0) && this.maze[x+1,y] != 0 && ! found)
then
        temp ← new Simpul(x,y,this.maze)
        temp.Append(x,y)
        enqueueResultInitHome(x + 1, y, ref temp, ref found)

    if (this.y-1 >= 0 && this.maze[x,y-1] != 0 && ! found) then
        temp ← new Simpul(x,y,this.maze)
        temp.Append(x,y)
        enqueueResultInitHome(x, y - 1, ref temp, ref found)

```

```

if (this.x-1 >= 0 && this.maze[x-1,y] != 0 && ! found) then
    temp ← new Simpul(x,y,this.maze)
    temp.Append(x,y);
    enqueueResultInitHome(x - 1, y, ref temp, ref found);

init():
    countVisited ← countVisited + 1
    initialize coorMap as an empty queue
    found ← false

    if (!tempProgress.isVisited(x,y))
        tempProgress.Append(x,y)
        tempProgress.visit(x,y)

        if (this.y+1 < this.maze.GetLength(1) && this.maze[x,y+1] != 0 && ! found
&& this.visitedMaze[x,y+1] != -1) then
            this.visitedMaze[x,y+1] ← -1
            temp ← new Simpul(x,y,this.maze)
            temp.Append(x,y)
            enqueueResultInit(x, y + 1, ref temp, ref found)

        if (this.x+1 < this.maze.GetLength(0) && this.maze[x+1,y] != 0 && !found
&& this.visitedMaze[x+1,y] != -1) then
            this.visitedMaze[x+1,y] ← -1
            temp ← new Simpul(x,y,this.maze)
            temp.Append(x,y)
            enqueueResultInit(x + 1, y, ref temp, ref found)

        if (this.y-1 >= 0 && this.maze[x,y-1] != 0 && !found &&
this.visitedMaze[x,y-1] != -1) then
            this.visitedMaze[x,y-1] ← -1
            temp ← new Simpul(x,y,this.maze)
            temp.Append(x,y)
            enqueueResultInit(x, y - 1, ref temp, ref found)

        if (this.x-1 >= 0 && this.maze[x-1,y] != 0 && ! found &&
this.visitedMaze[x-1,y] != -1) then
            this.visitedMaze[x-1,y] ← -1
            temp ← new Simpul(x,y,this.maze)
            temp.Append(x,y)
            enqueueResultInit(x - 1, y, ref temp, ref found)

isDone(int[,] arr):
    for i traversal [0 to arr's length on axis 0]:
        for j traversal [0 to arr's length on axis 1]:
            if (arr[i,j] == 3) then
                → false
            → true

findPath():
    stopLoop ← false
    while true then
        tempPath ← coorMap.Dequeue()
        tempCoor ← tempPath.getEl(tempPath.getIdx())

        if (canTravel(tempCoor[0],tempCoor[1]+1) &&

```

```

this.visitedMaze[tempCoor[0],tempCoor[1]+1] != -1) then
    this.visitedMaze[tempCoor[0],tempCoor[1]+1] ← -1
    append ← new Simpul(tempPath)
    processPath(tempCoor[0],tempCoor[1]+1, ref tempPath, ref append,
ref stopLoop)
        if (stopLoop) then
            break
        coorMap.Enqueue (append)
        countVisited ← countVisited + 1

        if (canTravel(tempCoor[0]+1,tempCoor[1]) &&
this.visitedMaze[tempCoor[0]+1,tempCoor[1]] != -1) then
            this.visitedMaze[tempCoor[0]+1,tempCoor[1]] ← -1
            append ← new Simpul(tempPath)
            processPath(tempCoor[0]+1,tempCoor[1], ref tempPath, ref append,
ref stopLoop)
                if (stopLoop) then
                    break
                coorMap.Enqueue (append)
                countVisited ← countVisited + 1

        if (canTravel(tempCoor[0],tempCoor[1]-1) &&
this.visitedMaze[tempCoor[0],tempCoor[1]-1] != -1) then
            this.visitedMaze[tempCoor[0],tempCoor[1]-1] ← -1
            append ← new Simpul(tempPath)
            processPath(tempCoor[0],tempCoor[1]-1, ref tempPath, ref append,
ref stopLoop)
                if (stopLoop) then
                    break
                coorMap.Enqueue (append)
                countVisited ← countVisited + 1

        if (canTravel(tempCoor[0]-1,tempCoor[1]) &&
this.visitedMaze[tempCoor[0]-1,tempCoor[1]] != -1) then
            this.visitedMaze[tempCoor[0]-1,tempCoor[1]] ← -1
            append ← new Simpul(tempPath)
            processPath(tempCoor[0]-1,tempCoor[1], ref tempPath, ref append,
ref stopLoop)
                if (stopLoop) then
                    break
                coorMap.Enqueue (append)
                countVisited ← countVisited + 1

finalTSP():
    finalSearch()
    goHome ← true
    tempProgress ← new Simpul(x,y,maze)
    this.flag ← true
    initHome()
    findPath()
    progress.Enqueue (tempProgress)
    res ← new Simpul(x,y,maze)
    res.unite(result)

finalSearch():
    tempCount ← 0

```

```

tempProgress ← new Simpul(x,y,maze)
this.flag ← true
while(tempCount < count) then
    tempProgress ← new Simpul(x,y,maze)
    init()
    if (this.flag) then
        findPath()
    progress.Enqueue(tempProgress)
    this.flag ← true
    tempCount ← tempCount + 1
    for i traversal [0 to visitedMaze's length on axis 0]:
        for j traversal [0 to visitedMaze's length on axis 1]:
            if (visitedMaze[i,j] == -1) then
                visitedMaze[i,j] ← 2

res ← new Simpul(x,y,maze)
res.unite(result)
res.displayCoord()

enqueueResultInitHome(int a, int b, ref Simpul temp, ref bool found):
    temp.Append(a, b)
    if (!this.tempProgress.isVisitedHome(a, b)) then
        this.tempProgress.Append(a, b)
    this.tempProgress.visitHome(a, b)
    if (temp.isHome(a, b)) then
        found ← true
        this.result.Enqueue(temp)
        this.flag ← false
        this.coorMap.Clear()
    this.coorMap.Enqueue(temp);

enqueueResultInit(int a, int b, ref Simpul temp, ref bool found):
    temp.Append(a, b)
    if (!this.tempProgress.isVisited(a, b)) then
        this.tempProgress.Append(a, b)
    this.tempProgress.visit(a, b)
    if (temp.isTreasure(a, b)) then
        temp.pickTreasure(a, b)
        found ← true
        this.result.Enqueue(temp)
        this.flag ← false
        this.x ← a
        this.y ← b
        this.maze ← copy of maze
        this.coorMap.Clear()
    this.coorMap.Enqueue(temp)
    this.countVisited ← this.countVisited + 1

processPath(int x, int y, ref Simpul tempPath, ref Simpul append, ref bool
stopLoop):
    if (goHome) then
        if (!tempProgress.isVisitedHome(x, y)) then
            this.tempProgress.Append(x, y)
            tempProgress.visitHome(x, y)
            append.Append(x, y)
            if (append.isHome(x, y)) then

```

```

        result.Enqueue(append)
        stopLoop ← true
    else
        if (!tempProgress.isVisited(x, y)) then
            this.tempProgress.Append(x, y)
        tempProgress.visit(x, y)
        append.Append(x, y)
        if (append.isTreasure(x, y)) then
            append.pickTreasure(x, y)
            result.Enqueue(append)
        this.coorMap.Clear()
        this.x ← x
        this.y ← y
        this.maze ← copy of maze
        stopLoop ← true

```

4.2.3. PseudoCode DepthFirst Search

```

DFS:
visited: stack
visitedHome: stack
result: stack
count: int
countSteps: int
totalTreasure: int
countVisited: int
maze: arr of (int,int)
visitedMaze: arr of (int,int)
x : int
y : int
DFS(int x, int y, int total, int[,] maze):
    initialize visited as an empty stack
    initialize visitedHome as an empty stack
    initialize result as an empty stack
    x ← x
    y ← y
    totalTreasure ← total
    maze ← maze
    visitedMaze ← copy of maze
    count ← 0
    countSteps ← 0
    countVisited ← 0

findHome method:
    first ← new Simpul object from top of visited stack
    visitedMaze[first.getX(), first.getY()] ← -2
    first ← visitedHome stack
    first ← result stack
    valid ← true

    while true then
        valid ← true
        top ← new Simpul object from top of visitedHome stack

```

```

        if (top = home) then
            add top back to visitedHome stack
            break

        if top.canGoRight and valid then
            if visitedMaze[top.getX(), top.getY() + 1] is not -2 then
                top.canGoRight ← false
                temp←new Simpul object at (top.getX(),top.getY()+1) with maze
                add top to visitedHome stack
                add temp to visitedHome stack
                add temp to result stack
                visitedMaze[top.getX(), top.getY() + 1] ← -2
                valid ← false

        if top.canGoDown and valid then
            if visitedMaze[top.getX() + 1, top.getY()] is not -2 then
                top.canGoDown ← false
                temp ← new Simpul object at (top.getX() + 1, top.getY()) with
maze
                add top to visitedHome stack
                add temp to visitedHome stack
                add temp to result stack
                visitedMaze[top.getX() + 1, top.getY()] ← -2
                valid ← false

        if top.canGoLeft and valid then
            if visitedMaze[top.getX(), top.getY() - 1] is not -2 then
                top.canGoLeft ← false
                temp ← new Simpul object at (top.getX(), top.getY() - 1) with
maze
                add top to visitedHome stack
                add temp to visitedHome stack
                add temp to result stack
                visitedMaze[top.getX(), top.getY() - 1] ← -2
                valid ← false

        if top.canGoUp and valid then
            if visitedMaze[top.getX() - 1, top.getY()] is not -2 then
                top.canGoUp ← false
                temp = new Simpul object at (top.getX() - 1, top.getY()) with
maze
                add top to visitedHome stack
                add temp to visitedHome stack
                add temp to result stack
                visitedMaze[top.getX() - 1, top.getY()] ← -2
                valid ← false

        if valid then
            add top back to visitedHome stack
            top ← new Simpul object from top of visitedHome stack
            add visitedHome.Peek() to result stack

findPath():
    first← new Simpul(this.x, this.y, maze)
    visitedMaze[x, y] ← -1
    visited.Push(first)
    result.Push(first)

```

```

top ← new Simpul(visited.Pop())
while (true) do
    valid ← true
    if (maze[top.getX(), top.getY()] = 3) then
        Count ← count + 1
        maze[top.getX(), top.getY()] ← 2
    if (count = totalTreasure) then
        visited.Push(top)
        break
    if (top.canGoRight && valid) then
        if (visitedMaze[top.getX(), top.getY() + 1] != -1):
            top.canGoRight ← false
            temp ← new Simpul(top.getX(), top.getY() + 1, maze)
            visited.Push(top)
            visited.Push(temp)
            result.Push(temp)
            visitedMaze[top.getX(), top.getY() + 1] ← -1
            valid ← false
    if (top.canGoDown && valid) then
        if (visitedMaze[top.getX() + 1, top.getY()] != -1) then
            top.canGoDown ← false
            temp ← new Simpul(top.getX() + 1, top.getY(), maze)
            visited.Push(top)
            visited.Push(temp)
            result.Push(temp)
            visitedMaze[top.getX() + 1, top.getY()] ← -1
            valid ← false
    if (top.canGoLeft && valid) then
        if (visitedMaze[top.getX(), top.getY() - 1] != -1) then
            top.canGoLeft ← false
            temp ← new Simpul(top.getX(), top.getY() - 1, maze)
            visited.Push(top)
            visited.Push(temp)
            result.Push(temp)
            visitedMaze[top.getX(), top.getY() - 1] ← -1
            valid ← false
    if (top.canGoUp && valid) then
        if (visitedMaze[top.getX() - 1, top.getY()] != -1) then
            top.canGoUp ← false
            temp = new Simpul(top.getX() - 1, top.getY(), maze)
            visited.Push(top)
            visited.Push(temp)
            result.Push(temp)
            visitedMaze[top.getX() - 1, top.getY()] ← -1
            valid ← false
    if (valid) then
        visited.Push(top)
        top = visited.Pop()
        result.Push(visited.Peek())
        countSteps←countSteps+1
displaySimpul:
    output x + " " + y

getMaze:
    -> maze

```

```

pickTreasure(x, y):
    maze[x, y] ← 2

displayPath:
    copy ← Stack(result)
    for value in copy:
        value.displaySimpul()

displaySteps:
    output countSteps

displayVisited:
    countVisited ← 0
    for i traversal [0 to visitedMaze's length on axis 0]:
        for j traversal [0 to visitedMaze's length on axis 1]:
            if visitedMaze[i, j] = -1 then
                countVisited ← countVisited + 1
    output countVisited

getResult:
    → result

getRoute:
    ret ← ""
    copy ← Stack(result)
    first ← true
    q ← copy.Pop()
    while copy's count > 0 do
        p ← q
        if first then
            first ← false
        else
            q ← copy.Pop()
        px ← p.getX()
        py ← p.getY()
        x ← q.getX()
        y ← q.getY()
        if x - px = 1 then
            ret ← ret + "D"
        elseif x - px = -1 then
            ret ← ret + "U"
        elseif y - py = 1 then
            ret ← ret + "R"
        elseif y - py = -1 then
            ret ← ret + "L"
    → ret

getCountVisited:
    → result's count

getStep:
    → result's count - 1

```

4.2. Struktur Data

4.2.1. Kelas Simpul

Atribut	Deskripsi
private int x	Atribut ini digunakan untuk menandai titik koordinat X dari titik awal
private int y	Atribut ini digunakan untuk menandai titik koordinat Y dari titik awal
public bool canGoLeft	Atribut ini berupa true atau false. Jika proses dapat jalan ke kiri, maka atribut ini bernilai true
public bool canGoRight	Atribut ini berupa true atau false. Jika proses dapat jalan ke kanan, maka atribut ini bernilai true
public bool canGoDown	Atribut ini berupa true atau false. Jika proses dapat jalan ke bawah, maka atribut ini bernilai true
public bool canGoUp	Atribut ini berupa true atau false. Jika proses dapat jalan ke atas, maka atribut ini bernilai true
int[,] maze;	Atribut ini berupa matriks yang menggambarkan sebuah maze
private List<(int,int)> arr	Atribut arr yang berisi list of tuple integer
private int idx	Atribut berupa banyak indeks suatu arr

Method	Deskripsi
public Simpul(int x, int y, int[,] maze)	Ctor simpul
public Simpul(Simpul other)	Cctor simpul
public void displayMaze()	Prosedur untuk menampilkan maze
public bool isVisited(int x, int y)	Fungsi bool yang mengidentifikasi suatu node apakah sudah pernah dilalui
public bool isVisitedHome(int x, int y)	Fungsi bool yang mengidentifikasi suatu node

	apakah sudah pernah dilalui, untuk TSP
public void visit(int x, int y)	Prosedur untuk menandai maze yang sudah di-visit dengan angka -1
public void visitHome(int x, int y)	Prosedur untuk menandai maze yang sudah di-visit dengan angka -2, untuk TSP
public List<(int,int)> getArr()	Fungsi getter yang mengembalikan arr
public void Append(int x, int y)	Prosedur untuk menambahkan suatu elemen x dan y pada arr
public void unite(Queue<Simpul> p)	Melakukan penggabungan untuk simpul yang ada pada queue sehingga menjadi satu simpul
public void displayCoord()	Menampilkan titik koordinat untuk simpul
public bool isTreasure(int x, int y)	Melakukan pengecekan apakah sebuah koordinat x dan y mengandung treasure
public bool isHome(int x, int y)	Melakukan pengecekan apakah titik x dan y merupakan home
public int getIdx()	Mengembalikan idx yang menjadi penunjuk untuk penelusuran DFS
public int getX()	Fungsi getter yang mengmbalikan suatu nilai X
public int getY()	Fungsi getter yang mengmbalikan suatu nilai Y
public int[] getEl(int id)	Mengembalikan array yang berisi variabel x dan y yang menunjukkan koordinat
public void displaySimpul()	Prosedur yang menampilkan simpul
public int[,] getMaze()	Fungsi getter yang mengembalikan maze
public void pickTreasure(int x, int y)	Mengganti titik x dan y pada peta menjadi jalur biasa

4.2.2. Kelas BFS

Atribut	Deskripsi

private bool flag	Boolean yang digunakan untuk membantu pencarian pada saat melakukan BFS untuk tiap treasure
private Queue<Simpul> coorMap	Queue yang bersifat temporary dan digunakan untuk menyimpan jalur - jalur untuk mencapai treasure
private Queue<Simpul> result	Queue utama yang digunakan untuk menyimpan jalur yang menghubungkan pada treasure
private Queue<Simpul> progress	Queue yang berguna untuk melihat progress dari pencarian BFS
private Simpul tempProgress	Queue yang digunakan untuk membantu pengisian Queue progress
private int x	Variabel yang digunakan untuk menandai titik koordinat X dari titik awal
private int[,] maze	Matriks yang menggambarkan maze
private int[,] visitedMaze	Matriks yang menggambarkan maze namun sudah ditandai apabila terdapat koordinat yang sudah dikunjungi.
private int y	Variabel yang digunakan untuk menandai titik koordinat Y dari titik awal
private int count	Variabel yang digunakan untuk mengetahui berapa banyak treasure yang sudah didapat
private int countSteps	Variabel untuk menghitung banyak langkah untuk menuju semua treasure
private int countVisited	Variabel untuk menghitung berapa kali jumlah pengecekan pada node
private bool goHome	Variabel untuk menentukan apakah program akan menjalankan skema TSP

Method	Deksripsi
public BFS(int x, int y, int[,] arr, int t)	Konstruktor dari BFS
public bool canTravel(int x, int y)	Mengembalikan true apabila koordinat x dan

	y mungkin untuk diakses
public void initHome()	Melakukan pengecekan untuk sisi adjason dari titik <i>treasure</i> akhir yang ditemukan untuk pencarian titik awal
public void init()	Melakukan pengecekan untuk sisi adjason dari titik awal untuk pencarian <i>treasure</i>
public bool isDone(int[,] arr)	Mengembalikan true apabila semua treasure sudah habis
public void findPath()	Mencari jalur untuk mencapai treasure hingga mendapatkannya
public void finalTSP()	Melakukan pencarian titik awal
public void finalSearch()	Melakukan pencarian dengan gabungan antara findPath dan init dengan tujuan agar mencari Treasure satu per satu
public void enqueueResultInitHome(int a, int b, ref Simpul temp, ref bool found)	Proses pencatatan jalur yang diperiksa apabila valid untuk pencarian titik awal
public void enqueueResultInit(int a, int b, ref Simpul temp, ref bool found)	Proses pencatatan jalur yang diperiksa apabila valid untuk pencarian <i>treasure</i>
public void processPath(int x, int y, ref Simpul tempPath, ref Simpul append, ref bool stopLoop)	Proses pemeriksaan suatu titik apakah titik tersebut valid untuk dilewati atau tidak
public void displayPath()	Menampilkan jalur dari result yang didapat
public Queue<Simpul> getResult()	Mengembalikan atribut result
public Queue<Simpul> getProgress()	Mengembalikan atribut progress
public int getStep()	Mengembalikan atribut countsteps
public String getRoute()	Mendapatkan rute berbentuk L, R, U, B dari setiap jalur untuk mencapai treasure
public int getCountVisited()	Mengembalikan atribut countVisited

4.2.3. Kelas DFS

Atribut	Deskripsi

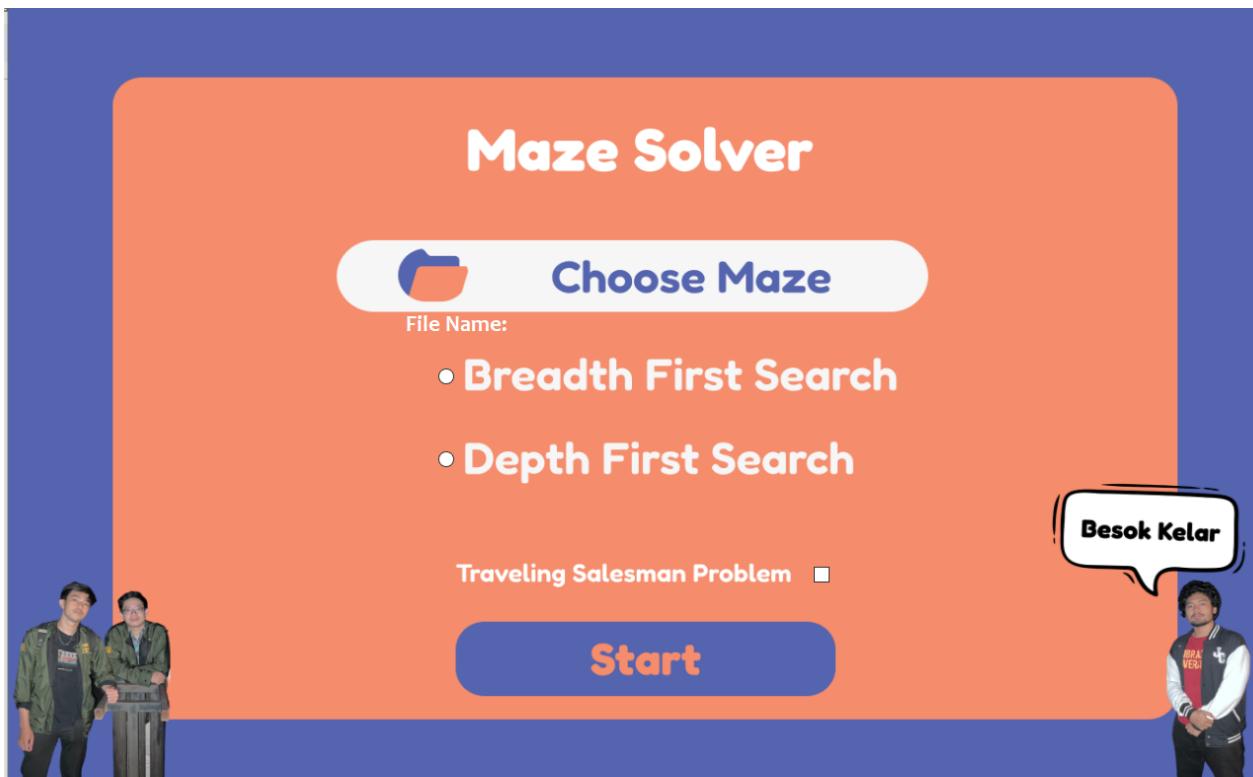
private Stack<Simpul> visited	<i>Stack</i> untuk menyimpan rute yang sedang dikunjungi untuk melakukan pencarian seluruh <i>treasure</i>
private Stack<Simpul> visitedHome	<i>Stack</i> untuk menyimpan rute yang sedang dikunjungi untuk melakukan pencarian kembali ke titik awal
private Stack<Simpul> result	<i>Stack</i> untuk mencatat rute yang dihasilkan
private int count	Jumlah <i>treasure</i> yang sudah ditemukan oleh program
private int countSteps	Jumlah pemeriksaan titik
private int totalTreasure	Jumlah <i>treasure</i> yang ada di dalam <i>maze</i>
private int countVisited	Jumlah titik yang sudah dikunjungi
private int[,] maze	Matriks
private int[,] visitedMaze	Matriks
private int x	Absis dari titik awal
private int y	Oordinat dari titik awal

Method	Deksripsi
public DFS(int x, int y, int total, int[,] maze)	Konstruksi yang menerima parameter matriks <i>maze</i> , jumlah <i>treasure</i> , dan absis beserta oordinat dari titik awal
Public void findHome()	Pencarian jalur kembali ke titik awal dari titik <i>treasure</i> terakhir yang ditemukan
Public void findPath()	Pencarian jalur ke seluruh <i>treasure</i> dalam <i>maze</i>
Public void pushResult(int x, int y, Simpul top)	Proses pencatatan jalur yang diperiksa apabila valid untuk pencarian <i>treasure</i>
Public void pushResultHome(int x, int y, Simpul top)	Proses pencatatan jalur yang diperiksa apabila valid untuk pencarian titik awal
public void displayPath()	Menampilkan hasil akhir jalur yang

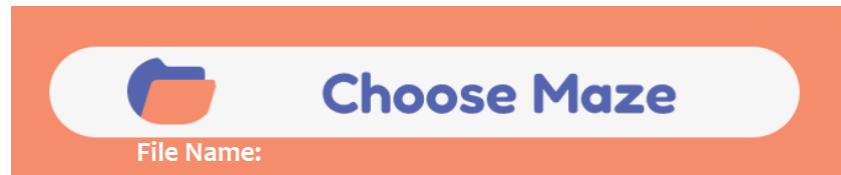
	ditemukan
public void displaySteps()	Menampilkan jumlah dari countSteps
public void displayVisited()	Menampilkan jumlah dari countVisited
public Stack<Simpul> getResult()	Fungsi yang mengembalikan suatu stack berisi result
public String getRoute()	Menampilkan rute untuk mengambil semua treasure yang berbentuk aksi seperti L, R, U, D
public int getCountVisited()	Mengembalikan atribut countVisited
public int getStep()	Mengembalikan atribut countsteps

4.3. Cara Penggunaan Program

Halaman Utama



1. Masukkan file .txt yang sesuai dengan klik tombol choose maze.



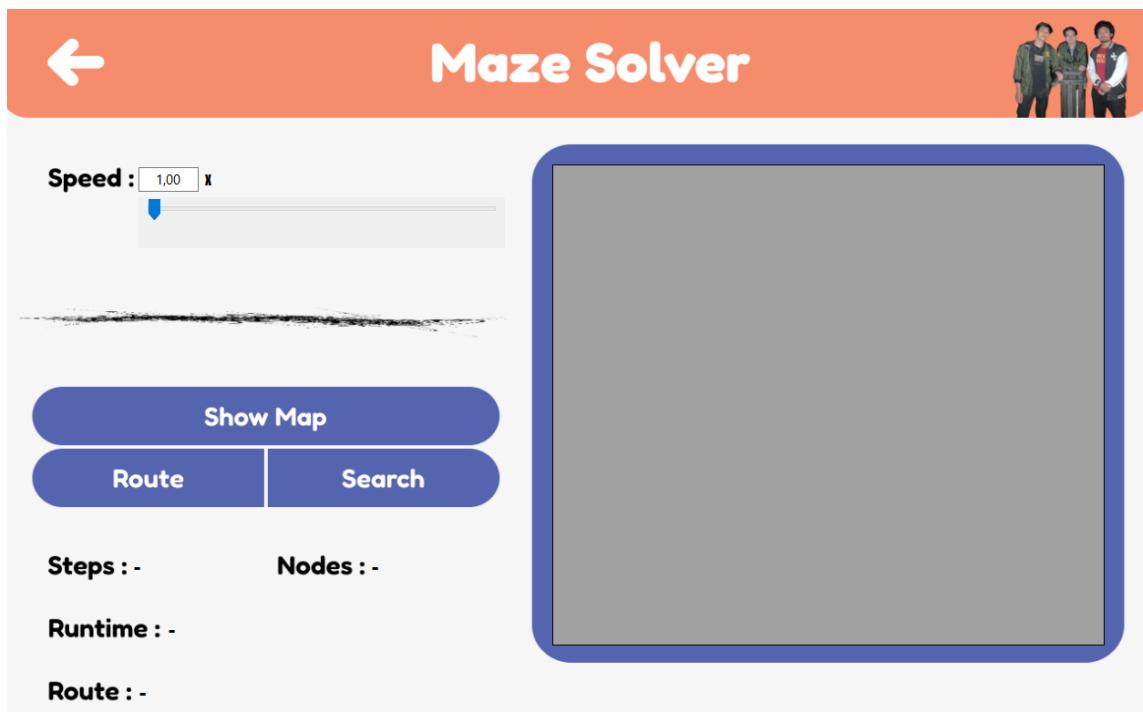
2. Pilih algoritma yang diinginkan dan pilih untuk melakukan tsp atau tidak



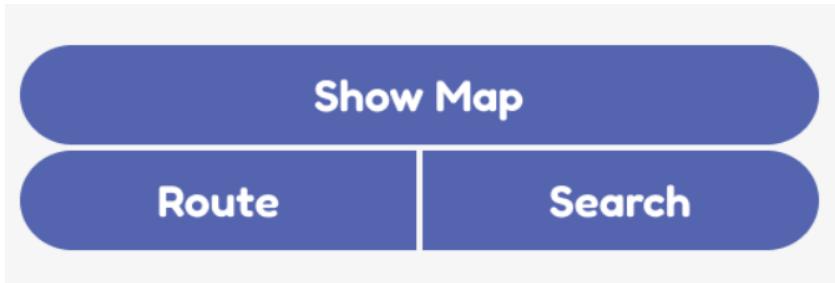
3. Klik tombol search



4. Anda akan diarahkan pada halaman kedua



5. Klik show map untuk menampilkan visualisasi peta. Untuk mendapatkan rute klik “route” dan untuk mendapatkan progress klik “search”.



6. Anda juga dapat mengubah kecepatan dengan menggeser slider diatas



4.6. Hasil Pengujian

4.6.1. Pencarian rute BFS Test Case 1

Speed : 10,00

Show Map

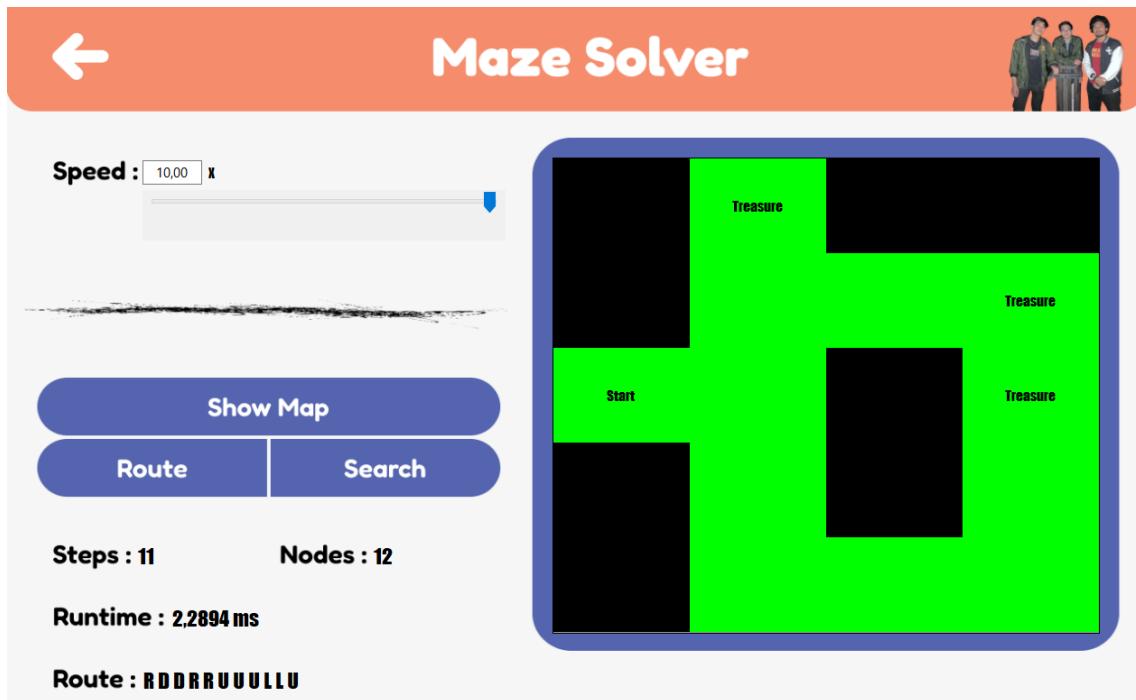
Route | Search

Steps : 7 Nodes : 14

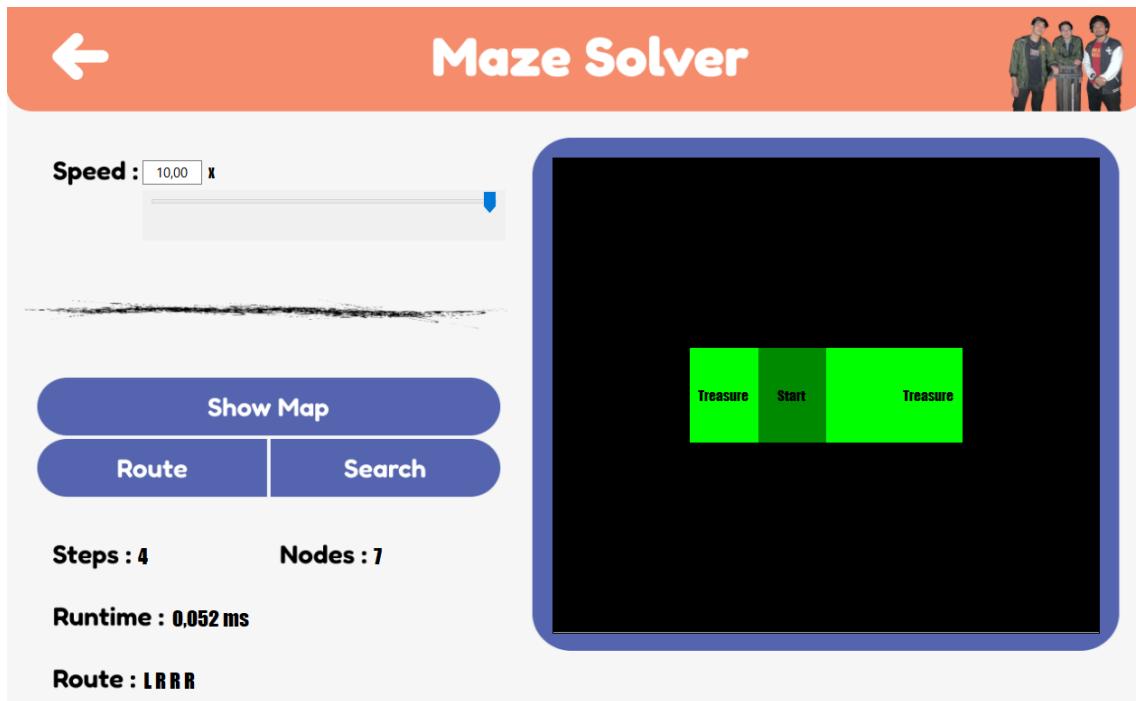
Runtime : 0,0431ms

Route : RUUDRRD

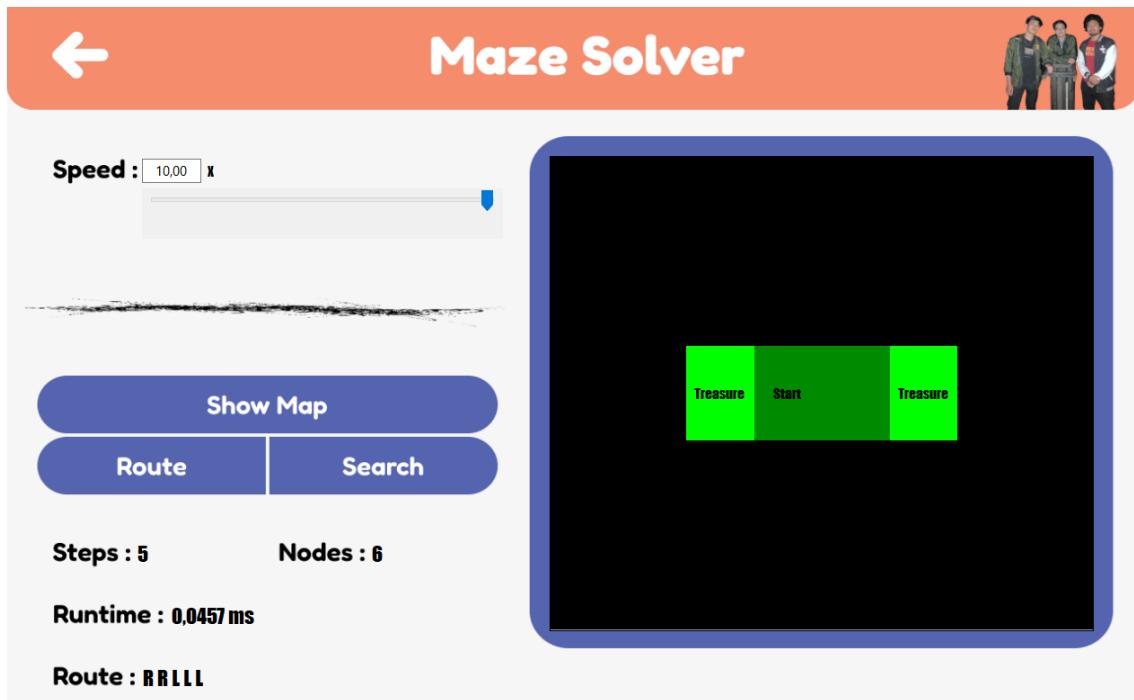
4.6.2. Pencarian rute DFS Test Case 1



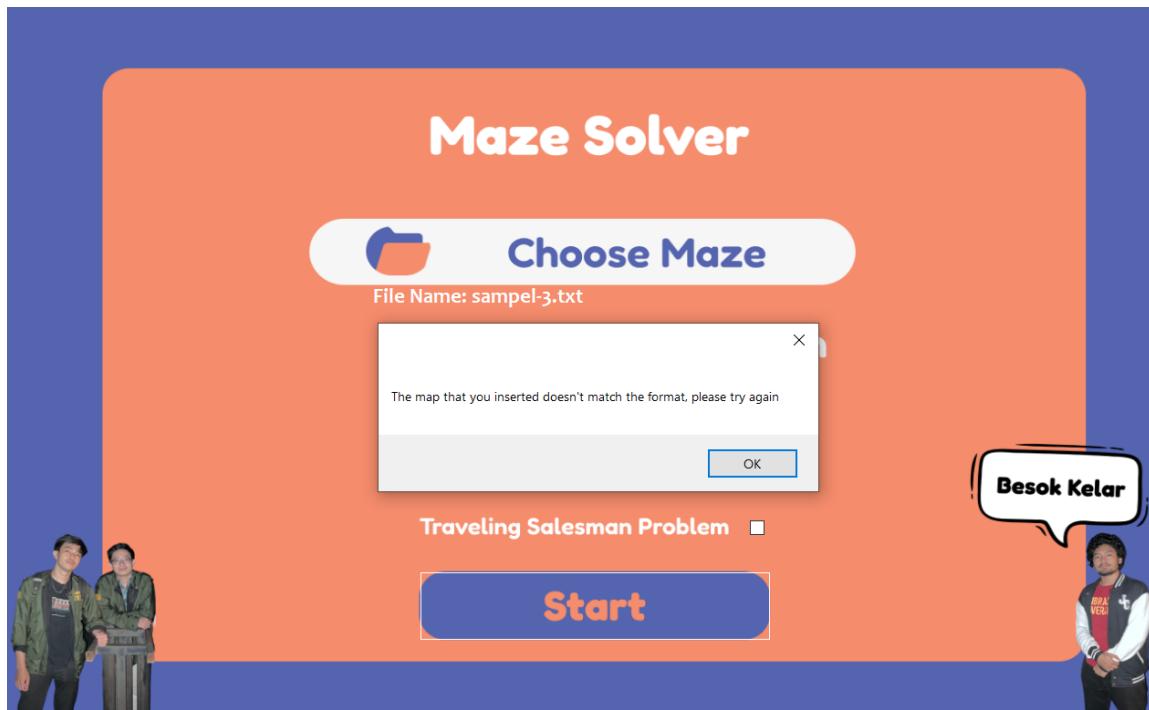
4.6.3. Pencarian rute BFS Test Case 2



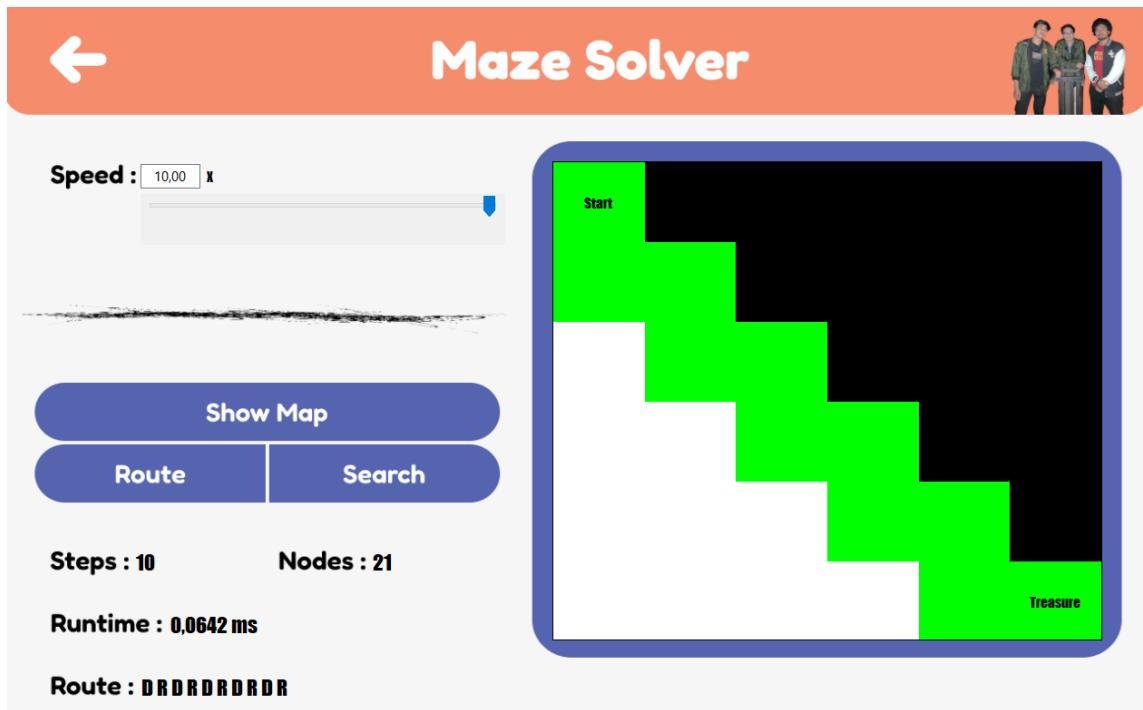
4.6.4. Pencarian rute DFS Test Case 2



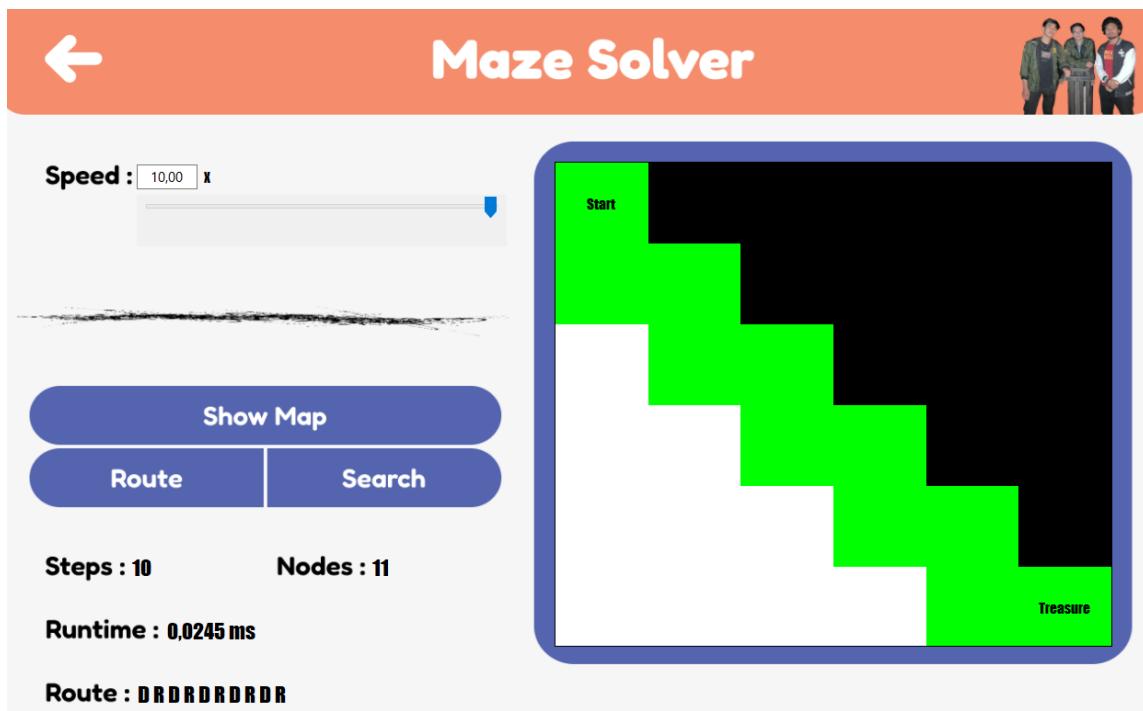
4.6.5. Pencarian rute BFS/DFS Test Case 3



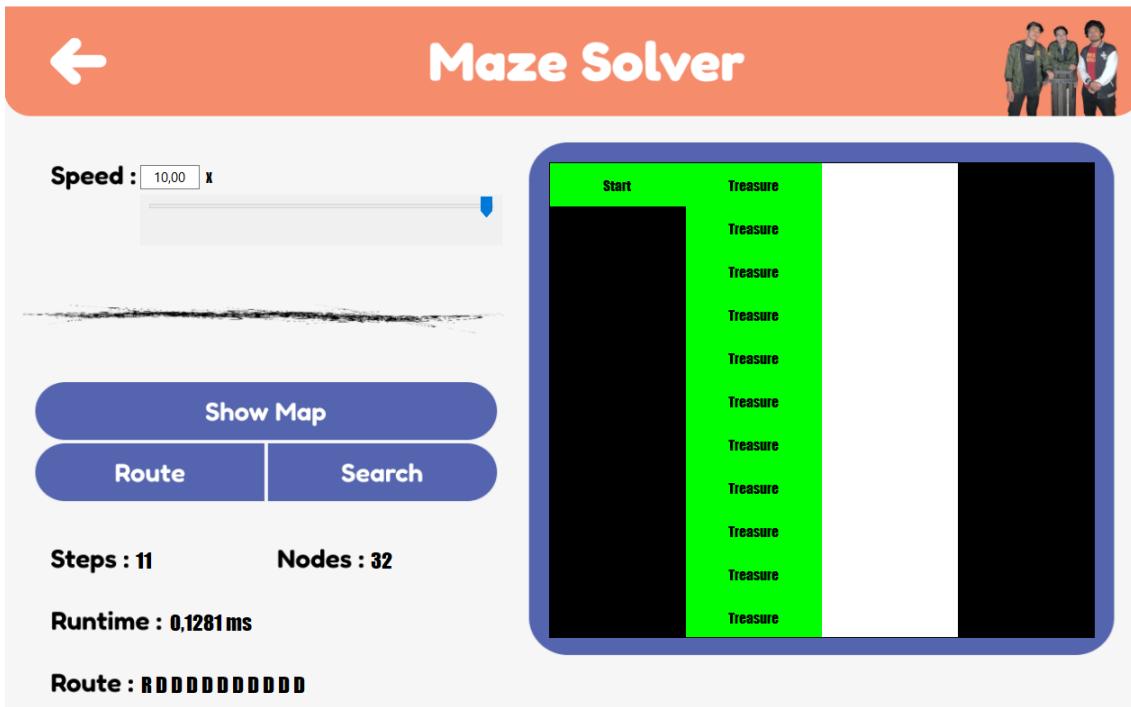
4.6.6. Pencarian rute BFS Test Case 4



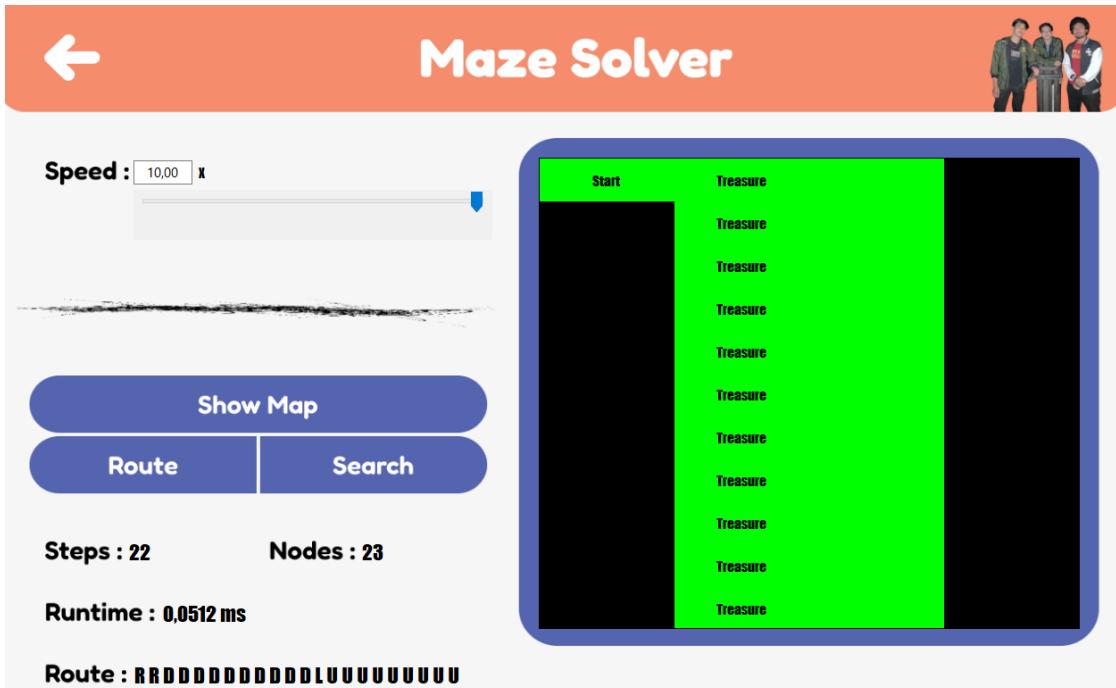
4.6.7. Pencarian rute DFS Test Case 4



4.6.8. Pencarian rute BFS Test Case 5



4.6.9. Pencarian rute DFS Test Case 5



4.6.10. Pencarian rute BFS TSP Test Case 1 (BONUS)

Maze Solver

←

Speed : 10,00 x

Show Map

Route | Search

Steps : 12 Nodes : 25

Runtime : 0,2445 ms

Route : RUUDRRRDULLDL

The interface shows a 4x4 grid with a blue border. The grid contains several black squares representing walls. There are three green squares labeled "Treasure" and one green square labeled "Start". The path found by the BFS algorithm is highlighted in green, starting from the "Start" square and ending at the first "Treasure" square. The path consists of the sequence: R, U, U, D, R, R, R, D, U, L, L, D, L, L, U, L.

4.6.11. Pencarian rute DFS TSP Test Case 1 (BONUS)

Maze Solver

←

Speed : 10,00 x

Show Map

Route | Search

Steps : 23 Nodes : 24

Runtime : 3,3914 ms

Route : RDDRRRUUULLLUUDRRDDDLUUL

The interface shows a 4x4 grid with a blue border. The grid contains several black squares representing walls. There are three green squares labeled "Treasure" and one green square labeled "Start". The path found by the DFS algorithm is highlighted in green, starting from the "Start" square and ending at the first "Treasure" square. The path consists of the sequence: R, D, D, R, R, R, U, U, U, L, L, L, L, U, U, D, R, R, D, D, D, L, L, U, U, L.

4.7. Analisis

BFS (Breadth-First Search) adalah algoritma yang mencari jalan keluar dari labirin dengan menjelajahi semua kemungkinan langkah di level yang sama sebelum pindah ke level berikutnya. Dalam implementasinya, BFS menggunakan struktur data baris untuk menyimpan node yang belum diperiksa, sehingga BFS dapat memeriksa node secara sistematis dan menjamin untuk menemukan solusi terpendek.

DFS (Depth-First Search) adalah algoritma yang menemukan jalan keluar dari labirin dengan menjelajahi setiap cabang sampai ke jalan buntu, kemudian kembali ke node sebelumnya dan menjelajahi cabang lainnya. Dalam sebuah aplikasi, DFS menggunakan struktur data tumpukan untuk menyimpan node yang tidak diperiksa, sehingga DFS dapat menjelajahi node secara sistematis, tetapi tidak dijamin untuk menemukan solusi terpendek.

Keuntungan BFS adalah menjamin solusi terpendek karena memeriksa semua kemungkinan pergerakan pada level yang sama sebelum pindah ke level berikutnya. Namun kerugiannya adalah jika solusinya jauh dari node aslinya, membutuhkan banyak memori dan waktu eksekusi yang lebih lama. Di sisi lain, keuntungan dari DFS adalah membutuhkan lebih sedikit memori dan biasanya lebih cepat daripada BFS untuk menemukan solusi yang lebih dekat ke node awal. Namun, sisi negatifnya adalah tidak dijamin untuk menemukan solusi terpendek dan dapat berputar jika tidak diterapkan dengan benar. Dikarenakan kelebihan dan kekurangan masing-masing algoritma, pemilihan algoritma BFS atau DFS tergantung pada karakteristik masalah dan tujuan pencarian yang ingin dicapai. Jika Anda ingin mencari solusi terpendek, BFS lebih disarankan. Namun, jika Anda ingin mencari solusi dengan cepat dan solusinya tidak terlalu jauh, disarankan untuk menggunakan algoritma DFS, karena pencarian DFS tidak memeriksa node sebanyak BFS.

BAB V

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Algoritma Breadth First Search (BFS) dan Depth First Search (DFS) memiliki kelebihan dan kekurangannya masing - masing. BFS memiliki kelebihan karena dapat menemukan jalur yang mendekati jalur terpendek dari titik awal ke titik akhir, namun membutuhkan memori yang lebih besar untuk menyimpan queue. Sementara itu, DFS lebih hemat memori namun memiliki kekurangan karena jalur yang ditemukan tidak selalu efektif.

Dalam prakteknya, kita dapat memilih algoritma BFS atau DFS tergantung pada kebutuhan kita. Jika kita ingin menemukan jalur dengan mementingkan keefektifannya, maka BFS adalah pilihan yang lebih tepat. Namun jika kita ingin menghemat memori, maka DFS dapat menjadi alternatif yang lebih baik.

5.2. Saran

Untuk memperluas saran dalam makalah tentang penyelesaian maze dengan algoritma DFS dan BFS, selain melakukan analisis efisiensi dan perbandingan dengan algoritma lain, dapat juga dilakukan pengujian terhadap berbagai jenis maze dengan karakteristik yang berbeda, seperti kompleksitas dan kepadatan jalan. Selain itu, penting untuk mempertimbangkan faktor lain yang dapat mempengaruhi kinerja algoritma, seperti kecepatan prosesor dan jumlah memori yang tersedia.

Terkait dengan algoritma A* dan backtracking, dapat dilakukan perbandingan yang lebih detail dengan algoritma DFS dan BFS, sehingga pembaca dapat mengetahui kelebihan dan kekurangan dari masing-masing algoritma dalam penyelesaian maze. Dalam makalah tersebut, dapat pula dibahas penggunaan algoritma yang lebih kompleks, seperti algoritma genetika, yang memiliki kemampuan untuk menemukan solusi yang lebih optimal dalam jangka panjang.

5.3. Refleksi

Algoritma Breadth First Search (BFS) dan Depth First Search (DFS) memiliki kelebihan dan kekurangan masing - masing. BFS dapat menemukan jalur yang mendekati jalur terpendek dari titik awal ke titik akhir, namun membutuhkan memori yang lebih besar untuk menyimpan queue. Sementara itu, DFS lebih hemat memori namun memiliki kekurangan karena jalur yang

ditemukan tidak selalu efektif. Dalam prakteknya, kita dapat memilih algoritma BFS atau DFS tergantung pada kebutuhan kita. Jika kita ingin menemukan jalur dengan mementingkan keefektifannya, maka BFS adalah pilihan yang lebih tepat. Namun jika kita ingin menghemat memori, maka DFS dapat menjadi alternatif yang lebih baik. Oleh karena itu, pemilihan algoritma yang tepat sangat bergantung pada kebutuhan dan karakteristik dari masalah yang ingin diselesaikan.

5.4. Tanggapan

Tugas pembuatan algoritma untuk mengambil semua treasure di dalam maze dengan BFS dan DFS merupakan suatu tantangan yang menarik. Melalui tugas ini, kita dapat memperdalam pemahaman tentang kedua algoritma dan penerapannya dalam menyelesaikan masalah konkret. Selain itu, tugas ini juga dapat melatih kemampuan logika dan pemrograman, serta meningkatkan keterampilan dalam mengembangkan dan menguji algoritma. Meskipun bisa menjadi tantangan yang sulit, menyelesaikan tugas ini akan memberikan pengalaman berharga dan meningkatkan kemampuan kita sebagai seorang mahasiswa dalam bidang informatika.

REFERENSI

Munir, R. (2020). Graf: Konsep dan Algoritma. [PDF file]. Retrieved from
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf>

Munir, R. (2021). Breadth First Search dan Depth First Search. [PDF file]. Retrieved from
<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf>

Javatpoint. (n.d.). AI - Uninformed Search Algorithms. Retrieved from
<https://www.javatpoint.com/ai-uninformed-search-algorithms>

GeeksforGeeks. (n.d.). Search Algorithms in AI. Retrieved from
<https://www.geeksforgeeks.org/search-algorithms-in-ai/>

Smart Mobility Algorithms. (n.d.). Blind Search. Retrieved from
<https://smartmobilityalgorithms.github.io/book/content/GraphSearchAlgorithms/BlindSearch.html>

TAUTAN

Github Repository : https://github.com/kennypanjaitan/Tubes2_besok-kelar.git

Youtube :  Tubes STIMA 2 - Besok Kelar